Tuning Paxos for high-throughput with batching and pipelining

Nuno Santos, André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL)
Email: firstname.lastname@epfl.ch

***Abstract*—Paxos is probably the most popular state machine replication protocol. Two optimizations that can greatly improve its performance are batching and pipelining. Nevertheless, tuning these two optimizations to achieve optimal performance can be challenging, as their effectiveness depends on many parameters like the network latency and bandwidth, the speed of the nodes, and the properties of the application. We address this question, by first presenting an analytical model of the performance of Paxos that can be used to obtain values for tuning batching and pipelining. We then present experiments validating the model and investigating how these two optimizations interact in both a LAN and a WAN setting. The results show that although batching by itself is usually sufficient to maximize the throughput in a LAN environment, in a WAN it must be complemented with pipelining.**

I. Introduction

State machine replication is a technique commonly used by fault tolerant systems. This technique allows the replication of any service that can be implemented as a deterministic state machine, *i.e.*, where the state of the service is determined only by the initial state and the sequence of commands executed. Given such a service, we need a protocol ensuring that each replica of the service executes the requests received from the clients in the same order.

Paxos is probably the most popular of such protocols. It is designed for partially synchronous systems with benign faults. In Paxos, a distinguished process, the leader, receives the requests from the clients and establishes a total order, using a series of instances of an ordering protocol.

In the simplest Paxos variant, the leader orders one client request at a time. In general, this is very inefficient for two reasons. First, since ordering one request takes at least one network round-trip between the leader and the replicas, the throughput is bounded by $\frac{1}{2L}$ where L is the network latency. This dependency between throughput and latency is undesirable, as it severely limits the throughput in moderate to high latency networks. Second, if the request size is small, the fixed costs of executing an instance of the ordering protocol can become the dominant factor and quickly overload the CPU of the replicas.

In this paper, we study two optimizations to the basic Paxos protocol that address these limitations: batching and pipelining. *Batching* consists of packing several requests in a single instance of the ordering protocol. The main benefit is spreading the fixed per-instance costs over several requests, which results in a smaller per-request overhead and potentially in a higher throughput. Batching can easily be implemented on top of Paxos, as it does not require any changes to the ordering protocol. *Pipelining* [1] is an extension of the basic Paxos protocol where the leader initiates new instances of the ordering protocol before the previous ones have completed. This optimization is particularly effective when the network latency is high, as it allows the leader to pipeline several instances on the slow link.

Batching and pipelining are used by most replicated state machine implementations, as they usually provide performance gains between one and two orders of magnitude. Nevertheless, in order to achieve the best throughput, they must be carefully tuned. For batching, it is necessary to choose the bound on the size of batches, in order to strike a balance between the size of batches and how long the leader has to wait for client requests. For pipelining, it is necessary to set a limit on the number of instances that can be in execution simultaneously, as a value that is too high may lead to a significant degradation in performance due to the increased overhead of managing multiple instances. Moreover, the optimal choice for the bounds on the batch size and number of parallel instances depends on the properties of the system and of the application, mainly on process speed, bandwidth, latency, and size of client requests.

We begin by studying analytically what are the combinations of batch size and number of parallel instances that maximize throughput for a given system and workload. We express this relationship in terms of a function $w = f(S_{batch})$, where S_{batch} is a batch size and w is a number of parallel instances (also denoted by window size). This result can be used to tune batching and pipelining, for instance, by setting the bounds on the batch and window size to one of the optimal combinations, so that given enough load the system will reach maximum throughput. To obtain the relation above, we developed an analytical model for Paxos, which predicts several performance metrics, including the throughput of the system, the CPU and network utilization of an instance, as well as its wall-clock duration. We then present the results of an experimental study comparing batching and pipelining in two settings, one representing a WAN and the other a cluster. We show which gains are to be expected by using either of the optimizations alone or combined, the results showing that although in some situations batching by itself is enough, in many others it must be combined with parallel instances. We contrast these

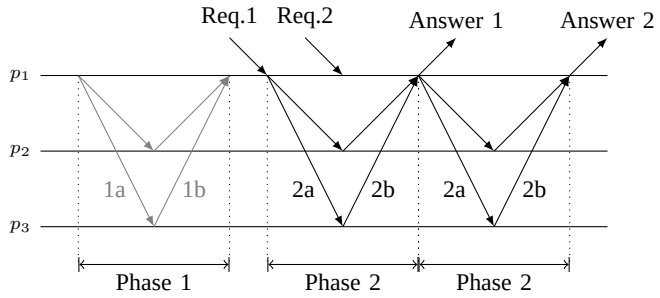
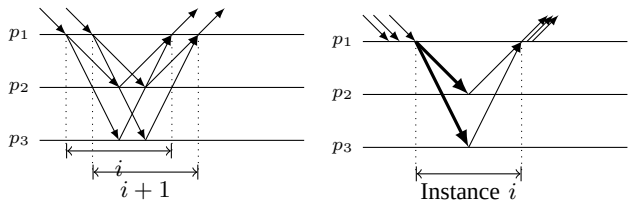


Figure 1. Basic MultiPaxos ($n = 3,\ f = 1$)



(a) Paxos with pipelining (b) Paxos with batching.

Figure 2. Paxos optimizations.

results with the prediction of our model, showing that the model is effective at predicting several performance metrics, including the throughput and optimal window size for a given batch size.

The rest of the paper is organized as follows. Section II provides the background for our work, describing in more detail the batching and pipelining optimizations in Paxos, Section III presents an analytical model of Paxos, Section IV presents the experimental evaluation of these two optimizations on a LAN (a cluster) and on a WAN, Section VI discusses the results of the paper, Section VII presents the related work, and Section VIII concludes the paper.

II. Background

Paxos, or more precisely MultiPaxos, is a state machine replication protocol, which at its core uses the Synod consensus algorithm [1]. If the replication degree is n, and f out of the n replicas may fail by crashing, the protocol requires $n \geq 2f + 1$. MultiPaxos can be seen as a *sequencer-based* atomic broadcast protocol [2], where the sequencer orders requests received from the clients. In the Paxos terminology, the sequencer is called *leader*. Paxos is usually described in terms of proposers, acceptors and learners, which are the roles each process can play. Here we ignore the different roles by assuming that every node plays all three roles.

For the purpose of the paper we describe only the relevant details of the Paxos protocol. Figure 1 shows the message pattern of Paxos for the case $n = 3,\ f = 1$. Once a process becomes leader (p_1 in Figure 1), it executes Phase 1 only once for all future instances. Afterwards, for each new request received from the clients, it only needs to execute Phase 2 (a request is ordered at the leader upon reception of enough Phase 2b messages), thereby saving two message delays per consensus instance. Therefore, in our analysis we ignore Phase 1 messages and use the term *instance* as an abbreviation for *one instance of Phase 2*.

In the simplest version of MultiPaxos, the leader proposes one request per instance and executes one instance at a time.

A. Pipelining

MultiPaxos can be extended to allow the leader to execute several instances in parallel [1]. In this case, when the leader receives a new request, it can start a new instance at once, even if there are other that are still undecided, as shown in Figure 2a.

Executing parallel instances *improves the utilization of resources* by pipelining the different instances. This optimization is especially effective in high-latency networks, as the leader might have to wait a long time to receive the Phase 2b messages.

The main drawback is that each instance requires additional resources from the system. If too many instances are started in parallel, they may overload the system, either by maxing out the leader's CPU or by causing network congestion, resulting in a more or less severe performance degradation. For this reason, the number of parallel instances that the leader is allowed to start is usually bounded. Choosing a good bound requires some careful analysis. If set too low, the network will be underutilized. If set too high, the system might become overloaded resulting in a severe performance degradation, as shown by the experiments in Section IV. The best value depends on many factors, including the network latency, the size of the requests, the speed of the replicas, and the expected workload.

B. Batching

Batching is a common optimization in communication systems, which generally provides large gains in performance [3]. It can also be applied to Paxos, as illustrated by Figure 2b. Instead of proposing one request per instance, the leader packs several requests in a single instance. Once the order of a batch is established, the order of the individual requests is decided by a deterministic rule applied to the request identifiers.

The gains of batching come from spreading the fixed costs of an instance over several requests, thereby decreasing the average per-request overhead. For each instance, the system performs several tasks that take a constant time regardless of the size of the proposal, or whose time increases only residually as the size of the proposal increases. These include interrupt handling and context switching as a result of reading and writing data to the network card, allocating buffers, updating the replicated log and the internal data structures, and executing the protocol logic. In [4], the authors show that the fixed costs of sending a packet over a Ethernet network are dominant for small packet sizes, and that for

larger packets the total processing time grows significantly slower than the packet size. In the case of Paxos, the fixed costs of an instance are an even larger fraction of the total costs because, in addition to processing individual messages, processes also have to execute the ordering algorithm. Additionally, batching decreases dramatically the cost of using stable storage, because a single stable storage access is enough to log the state of all requests in a batch.

Batching is fairly simple to implement in Paxos: the leader waits until having "enough" client requests and proposes them as a single proposal. The difficulty is deciding what is "enough". In general, the larger the batches, the bigger the gains in throughput. But in practice, there are several reasons to limit the size of a batch. First, the system may have physical limits on the maximum packet size (for instance, the maximum UDP packet size is 64KB). Second, larger batches take longer to build because the leader has to wait for more requests, possibly delaying the ones that are already waiting and increasing the average time to order each request. This is especially problematic with low load, as it may take a long time to form a large batch. Finally, a large value takes longer to transfer and process, further increasing the latency. Therefore, a batching policy must strike a balance between creating large batches (to improve throughput) and deciding when to stop waiting for additional requests and send the batch (to keep latency within acceptable bounds). This problem has been studied in the general context of communication protocols by [4]–[6]. In the rest of the paper, we study it in the context of Paxos, and analyze its interaction with the pipelining optimization.

III. Analytical model of Paxos performance

We consider the Paxos variant described in Section II with point-to-point communication. There are other variants of Paxos that use different communication schemes, like IP multicast and chained transmission in a ring [7]. We chose the basic variant for generality and simplicity, but this analysis can be easily adapted to other variants. We further assume full duplex links and that no other application is competing for bandwidth or CPU time.[1] Also for simplicity, we focus on the best case, that is, we do not consider message loss or failures. We also ignore mechanisms internal to a full implementation of Paxos, like failure detection. On a finely tuned system, these mechanisms should have a minimal impact on throughput.

Finally, we assume that execution within each process is sequential. The model can be extended to account for multi-core or SMP machines, but this is a non-trivial extension which, for the sake of simplicity, we do not explore here.

Symbol	Description
n	Number of replicas
B	Bandwidth
L	One way delay (latency)
S_{req}	Size of request
k	Number of requests in a batch
w	Number of parallel instances
S_{2a}	Size of a Phase 2a message (batch)
S_{2b}	Size of ack
S_{ans}	Size of answer sent to client
ϕ_{exec}	CPU-time used to execute a request
WND	Bound on maximum number of parallel instances (Configuration parameter)
BSZ	Bound on batch size (Configuration parameter)

Table I
NOTATION.

A. Quantitative analysis of Phase 2 of Paxos

Table I shows the parameters and the notation used in the rest of the paper. We focus on the two resources that are typically the bottleneck in a Paxos deployment, *i.e.*, the leader's CPU and its outgoing channel.

Our model takes as input the system parameters (n, B, L, and four constants defined later that model the speed of the nodes), the workload parameters (S_{req}, S_{ans} and ϕ_{exec}), and the batching level (k). From these parameters, the model characterizes how an instance utilizes the two critical resources, by determining the duration of an instance (wall-clock time), and the busy time of each resource, that is, the total time during which the resource is effectively used. With these two values, we can then determine the fraction of idle time of a resource, and predict how many additional parallel instances are needed to reach maximum utilization. The resource that reaches saturation with the lowest number of parallel instances is effectively the bottleneck, so it is this resource that determines the maximum number of parallel instances that can be executed in the system.

The model also provides estimations for the throughput and latency with a given configuration, which can be used to study how different batch sizes affect the performance and the optimal number of parallel instances for each batch size.

For simplicity, we assume that all requests are of similar size. Since the bulk of the Phase 2a message is the batch being proposed, in the following we use $S_{2a} = kS_{req} + c$ to denote the batch size, where c represents the protocol headers.

1) Network busy time: The outgoing network channel of the leader is busy for the time necessary to send all the data related to an instance, which consists of $n - 1$ Phase 2a messages, one to every other replica, and k answers to the clients.

Because of differences in topology, we consider the cases of a LAN and a WAN separately. On a LAN, the replicas are typically on the same network, so the effective bandwidth

[1]The presence of other applications can be modeled by adjusting the model parameters, to reflect the competition for network resources.

available between them is the bandwidth of the network. Therefore, the leader has a total bandwidth of B to use for all the messages it has to send, and we can compute the time the network is used for an instance as follows:

$$\phi_{inst}^{\text{LAN}} = ((n-1)S_{2a} + kS_{ans})/B$$

On a WAN environment, however, the replicas are in different data centers, so the connection between them is composed of a fast segment inside the replica's data center (bandwidth B_L), and of another comparatively slow segment between the different data centers (bandwidth B_W). Since usually $B_W \ll B_L$, in the following analysis we consider B_W to be the effective bandwidth between the replicas, ignoring B_L, *i.e.*, we take $B = B_W$. Moreover, while in LAN a replica has a total bandwidth of B to share among all other replicas, on a typical WAN topology each replica has a total of B_W bandwidth to every other replica. The reason is that the inter-data center section of the connection between the replicas will likely be different for each pair of replicas, so that after leaving the data center, the messages from a replica will follow independent paths to each other replica. Thus, contrary to the case of a LAN, every message sent by the leader uses a separate logical channel of bandwidth B. By the same reasoning, the messages from the leader to the clients also use separate channels. Since sending the answers to the client does not delay executing additional instances, the network bottleneck are the channels between the leader and the other replicas. Therefore, we get:

$$\phi_{inst}^{\text{WAN}} = S_{2a}/B$$

In both cases, the per request time is given by $\phi_{req}^{\text{NET}} = \phi_{inst}^{\text{NET}}/k$, where NET stands for either LAN or WAN. The maximum network throughput of instances and requests is given by $1/\phi_{inst}^{\text{NET}}$ and $1/\phi_{req}^{\text{NET}}$, respectively.

2) CPU time: During each instance, the leader uses the CPU to perform the following tasks: read the requests from the clients, prepare a batch containing k requests, serialize and send $n-1$ Phase 2a message, receive $n-1$ phase 2b messages, execute the requests and send the answers to the clients (in addition to executing the protocol logic whenever it receives a message).

These tasks can be divided in two categories: interaction with clients and with other replicas. The CPU time required to interact with clients depends mainly on the size of the requests (S_{req}) and the number of requests that must be read to fill a batch (k), while the interaction with replicas depends on the number of replicas (n) and the size of the batch (S_{2a}). Since these two interactions have distinct parameters, we model them by two functions: $\phi_{cli}(x)$ and $\phi_{rep}(x)$. The function $\phi_{cli}(x)$ represents the CPU time used by the leader to receive a request from a client and send back the corresponding answer, with x being the sum of the sizes of the request and the answer. Similarly, $\phi_{req}(x)$ is the CPU time used by the leader to interact with another replica, where x is the sum of the sizes of the Phase 2a and 2b messages. Both functions are linear, which models the well-known [4] behavior where the time to process a message consists of a constant plus a variable part, the later increasing linearly with the size of message[2]. The values of the parameters of these two functions must be determined experimentally for each system, as they depend both on the hardware used to run the replicas and on the implementation of Paxos. We show how to do so on Sections IV-A2 and IV-B2.

Therefore, we get the following for an instance CPU time:

$$\begin{aligned}\phi_{inst}^{\text{CPU}} =& k\phi_{cli}(S_{req} + S_{ans}) + \\ & (n-1)\phi_{rep}(S_{2a} + S_{2b}) + k\phi_{exec}\end{aligned}$$

The first term models the cost of receiving k requests from the clients and sending back the corresponding answers, the second term represents the cost of processing $n-1$ Phase 2a and 2b messages, finally, the last term is the cost of executing the k requests $k\phi_{exec}$.

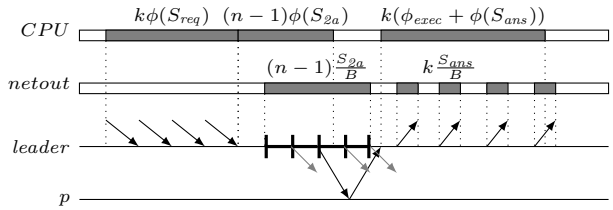
The time per request is given by $\phi_{req}^{\text{CPU}} = \phi_{inst}^{\text{CPU}}/k$. and the throughput in instances and request per seconds by $1/\phi_{inst}^{\text{CPU}}$ and $1/\phi_{req}^{\text{CPU}}$, respectively.

3) Wall-clock time: Estimating the wall-clock duration of an instance is more challenging than estimating the network and CPU utilization, because some operations that must complete for the instance to terminate are done in parallel. As an example, once the leader finishes sending $\lfloor n/2 \rfloor$ messages to the other replicas, the execution splits into two separate sequence of events. In one of them, the leader sends the remaining phase 2a messages. On the other, it waits for enough phase 2b messages to decide and start executing the requests. If after executing the first request in the batch, the leader did not finish sending all the Phase 2a messages, it may have to wait for the outgoing link to be free before sending the answers to the clients.

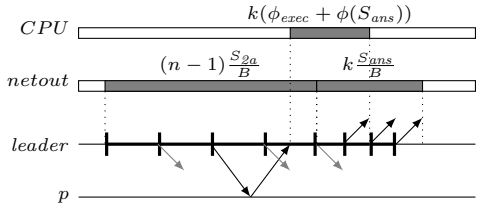
Therefore, the exact sequence of events that leads to completion depends on the workload and the characteristics of the system. In a fast LAN the wall-clock duration is likely to be limited by the CPU speed, while in high-latency WAN the latency is likely the dominant factor. Similarly, if the workload consists of large requests and answers, the bandwidth is more likely to be the bottleneck than the CPU or the latency.

Therefore we model the wall-clock time by considering three different cases, each corresponding to a different bottleneck: CPU, bandwidth or latency. For each case, we compute the duration of an instance, which gives us three formulas: T_{inst}^{CPU}, T_{inst}^{band} and T_{inst}^{lat}. The instance time is the

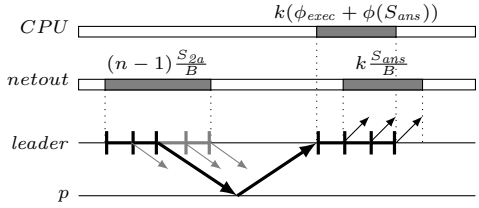
[2]We chose to use a single function to represent sending and receiving a pair of related messages, instead of one function per message type. Since the model is linear, this reduces the number of parameters that have to be estimated to half without losing any expressiveness.



(a) CPU is the bottleneck



(b) Bandwidth is the bottleneck



(c) Latency is the bottleneck

Figure 3. Utilization of the CPU and outgoing link of the leader during an instance.

maximum of the three:

$$T_{inst} = \max(T_{inst}^{\text{CPU}}, T_{inst}^{band}, T_{inst}^{lat}) \quad (1)$$

Once again, due to the differences in topology, we model the LAN and the WAN cases differently. For the LAN case, we have:

$$T_{inst}^{\text{CPU}} = \phi_{inst}^{\text{CPU}} + \lfloor n/2 \rfloor S_{2a}/2B \quad (2)$$
$$T_{inst}^{band} = ((n-1)S_{2a} + kS_{ans})/B \quad (3)$$
$$T_{inst}^{lat} = \lfloor n/2 \rfloor S_{2a}/B + 2L + k\phi_{exec} + kS_{ans}/B \quad (4)$$

Figure 3 illustrates the three cases. Each sub-figure represents an instance. The two lines at the bottom represent the leader and one other replica, the one whose Phase 2b message triggers the decision at the leader, and the two bars at the top represent the busy/idle periods of the CPU and of the outgoing link of the leader. The arrows above the leader line represent messages exchanged with the clients (their time-lines are not represented) and the arrows below are messages exchanged with the other replicas.

If the CPU is the bottleneck (Equation (2) and Figure 3a), the wall-clock time of an instance is dominated by the CPU time of this instance, which we have previously computed (formula ϕ_{inst}^{CPU} in Section III-A2). Additionally, the wall-clock time must also include the time during which the leader is sending the Phase 2a messages to other replicas, because its CPU will be partially idle as it waits for the answers. This difference between CPU and wall clock time increases with the size of the batch (See Figure 5). This idle time is represented by $\lfloor n/2 \rfloor S_{2a}/2B$.

If the bandwidth is the bottleneck (Equation (3) and Figure 3b), the wall-clock time of an instance is the total time needed by the leader to send all the messages of that instance through the outgoing channel, *i.e.*, $n-1$ Phase 2a messages and k answers.

Finally, if the latency is the bottleneck (Equation (4) and Figure 3c), the wall-clock time of an instance corresponds to the time needed to send the first $\lfloor n/2 \rfloor$ phase 2a messages to the replicas, plus the round-trip time required to receive enough Phase 2b messages from the replicas, followed by the execution time of the requests and the time to send the answers back to the clients.

For the WAN case, the formulas are as follow:

$$T_{inst}^{\text{CPU}} = \phi_{inst}^{\text{CPU}} + S_{2a}/B \quad (5)$$
$$T_{inst}^{band} = S_{2a}/B \quad (6)$$
$$T_{inst}^{lat} = S_{2a}/B + 2L + k\phi_{exec} \quad (7)$$

The difference is that messages can be send in parallel, because of the assumption that each pair of processes has exclusive bandwidth. Therefore, the time to send a message to the other replicas does not depend on n and sending the answers to the clients does not affect the duration of an instance (separate client-leader and leader-replica channels).

B. Maximizing resource utilization

If the leader's CPU and outgoing channel are not completely busy during an instance, then the leader can execute additional instances in parallel. The idle time of a resource R (CPU or outgoing link) is given by $T_{inst} - \phi_{inst}^{R}$ and the number of instances that a resource can sustain, w^R, is T_{inst}/ϕ_{inst}^{R}. From these, we can compute the maximum number of parallel instances that the system can sustain as:

$$w = \min(w^{\text{CPU}}, w^{\text{NET}}) \quad (8)$$

This value can be used as a guideline to configure batching and pipelining. In theory, setting the window size to any value equal or higher to this lower bound results in optimal throughput, but as shown by the experiments in Section IV-B, increasing the window size too much may result in congestion of the network or saturation of the CPU, and reduce performance. Therefore, the window size should be set to the lowest value suggested by the analytical model.

IV. Experimental Study

In this section we study the batching and pipelining optimizations from an experimental perspective, and validate the analytical model. Section IV-A shows the results in a Cluster and Section IV-B the results in a WAN environment

emulated using Emulab [8]. For each environment, we first present the experimental results. Then we determine the parameters of the model that represent the process speed (parameters of $\phi_{cli}(x)$ and $\phi_{rep}(x)$), and finally compare the predictions for the throughput and optimal window size of the model with the values obtained experimentally.

We performed the experiments using JPaxos [9], a full-feature implementation of Paxos in Java, which supports both batching and pipelining.

Implementing batching and pipelining in Paxos is fairly straightforward: batching has a trivial implementation and pipelining was described in the original Paxos paper [1]. To control these optimizations, *i.e.*, decide when to create a new batch and initiate a new instance, we use a simple algorithm with three parameters: WND, BSZ and Δ_B. The parameter WND is the maximum number of instances that can be executed in parallel, BSZ is the maximum batch size (in bytes), and Δ_B is the batch timeout. The timeout Δ_B is reset whenever the leader opens a new batch, which happens when it receives the first request that will go in the batch. The leader then waits until either it has enough requests to fill the batch or the timeout Δ_B expires. It then proposes the batch by starting a new instance as soon as the number of active instances is under WND. In the experiments we vary BSZ and WND while keeping Δ_B set to 50ms. This timeout has no impact on the results, as the load on the system is high enough for the leader to form a batch before the timeout expires.

We consider a system with three replicas. In order to stress the batching and pipelining mechanisms, all the experiments were performed with the system under high load. We used 900 clients in the cluster and 1200 in Emulab, which is enough for the leader to form new batches without having to wait for additional requests.

The replicated service keeps no state. It receives requests containing an array of S_{req} bytes and answers with an 8 bytes array. We chose a simple service as this puts the most stress on the replication mechanisms. JPaxos adds a header of 16 bytes per request and 4 bytes per batch of requests. The analytical results reported below take the protocol overhead in consideration.

All communication is done over TCP. We did not use IP multicast because it is not generally available in WAN-like topologies. Initially we considered UDP, but rejected it because in our tests it did not provide any performance advantage over TCP. TCP has the advantage of providing flow and congestion control, and of having no limits on message size, therefore saving us the tedious work of reimplementing these features. The replicas open the connections at startup and keep them open until the end of the run. Each data point in the plots corresponds to a 3 minutes run, excluding the first 10%. For clarity, the plots below do not include error bars for the 95% confidence interval, as the errors are usually very small.

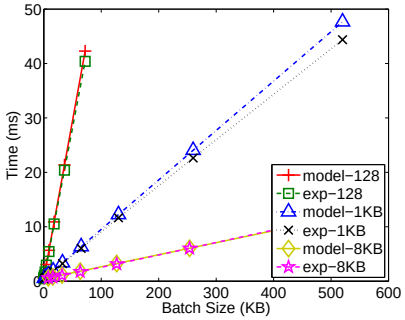


Figure 5. Experimental versus model results for the CPU time of an instance in the cluster. Model parameters: $\phi_{cli}(x) = 0.005x + 0.08, \phi_{rep}(x) = 0.0035x + 0.22$.

A. Cluster

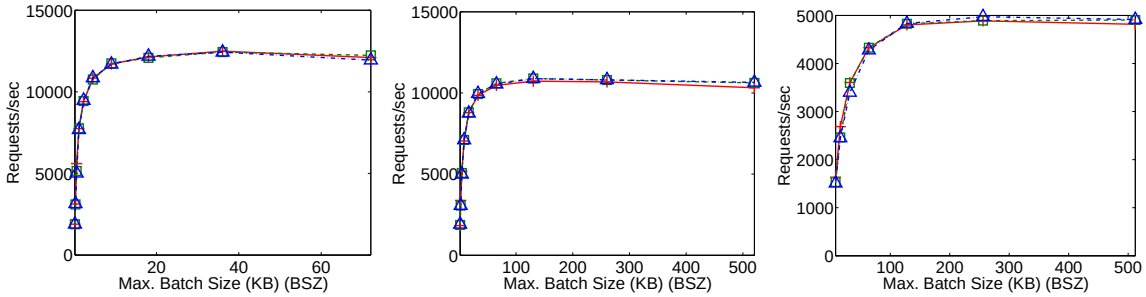
The following experiments were run on a cluster of Pentium 4 at 3GHz with 1GB memory connected by a Gigabit Ethernet. The effective bandwidth of a TCP stream between two nodes measured by `netperf` is 940 Mbit/s.

1) Experimental results: Figure 4 shows the request throughput as a function of batch size, for request sizes of 128 bytes, 1KB and 8KB, and for maximum window sizes of 1, 2 and 5.

Batching provides a major improvement in performance in all cases, ranging from an almost 10 times improvement with 128 bytes requests to a little over 4 times with 8KB requests. The batch size where the system reaches optimal throughput varies depending on the request size: around 10KB, 64KB and 128KB for request sizes of 128 bytes, 1KB and 8KB, respectively. On the other hand, increasing WND does not improve performance. During each run the average CPU utilization of the leader's CPU is above 90%, suggesting that the leader is CPU-bound and, therefore, is not able to execute additional instances.

The performance does not drop if the BSZ or WND are increased past their optimal values. This is a desirable behavior, because the system will perform optimally with a wide range of configuration parameters, making it easier to tune. As Section IV-B shows, this is not always the case.

2) Setting model parameters: To estimate the parameters ϕ_{cli} and ϕ_{rep} we used the Java Management interfaces (`ThreadMXBean`) to measure the total CPU time used by the leader process during a run. Dividing this value by the total number of instances executed during the run gives the average per-instance CPU time. To prevent the JVM warm-up period from skewing the results, we ignore the first 30 seconds of a run (for a total duration of 3 minutes). We repeat the measurements for several request and batch sizes, and then adjust the parameters of the model manually until the model's estimation for the CPU time (ϕ_{inst}^{CPU}) fits the training data. Figure 5 shows the training data together with the results of the model, for the final fit of $\phi_{cli}(x) = 0.005x + 0.08$ and $\phi_{rep}(x) = 0.0035x + 0.22$. The Figure shows that the CPU time measured experimentally



(a) $S_{req} = 128$ (b) $S_{req} = 1$KB (c) $S_{req} = 8$KB

Figure 4. Cluster. Experimental results: throughput as a function of batch size.

S_{2a}	ϕ_{req}^{CPU}	ϕ_{inst}^{CPU}	ϕ_{inst}^{NET}	T_{inst}	w^{CPU}	w^{NET}
128	0.52	0.52	0.00	0.52	1.00	211.73
256	0.30	0.60	0.00	0.60	1.00	124.18
512	0.19	0.77	0.01	0.77	1.00	79.52
1KB	0.14	1.09	0.02	1.10	1.00	56.97
2KB	0.11	1.75	0.04	1.76	1.01	45.63
4KB	0.10	3.06	0.08	3.07	1.01	39.95
8KB	0.09	5.67	0.15	5.71	1.01	37.11
16KB	0.09	10.90	0.31	10.98	1.01	35.68
32KB	0.08	21.36	0.62	21.51	1.01	34.97
64KB	0.08	42.28	1.23	42.58	1.01	34.62

Table II
ANALYTICAL RESULTS FOR CLUSTER, $S_{req} = 128$ (TIMES IN MILLISECONDS)

increases roughly linearly with the size of the batch, which validates our choice of a linear model.

3) Comparison of analytical and experimental results: All the analysis below is done with $\phi_{exec} = 0$, since the request execution time of the service used in the experiments is negligible (recall that the service simply answers with a 8 byte array).

Table II shows detailed results for the case $S_{req} = 128$, while Table III shows a summary of the analytical results for all request sizes and compares them with the experimental results.

With $S_{req} = 128$, the CPU time used by an instance (column ϕ_{inst}^{CPU}) is an order of magnitude larger than the network busy time (column ϕ_{inst}^{NET}). As a result, the wall-clock time of an instance (column T_{inst}) is dominated by the CPU time. Although the network could sustain many parallel instances (column w^{NET}), the CPU cannot sustain more than one (column w^{NET}) and therefore the system as a whole has no capacity to execute additional instances. The situation is similar for larger requests sizes (Tables IIIb and IIIc), although the CPU becomes less of a bottleneck as the size of the requests increases. A similar pattern occurs as the batch size increases, with the load shifting from the CPU to the network. But even with the largest messages tested, *i.e.*, $S_{req} = 8$KB and $S_{2a} = 512$KB, the CPU is still the bottleneck, being able to sustain only 1.17 parallel instances as compared to 1.63 of the network. Such a situation is

	Model			Experiments	
S_{2a}	w^{CPU}	w^{NET}	Max Thrp	w	Max Thrp
128	**1**	211.73	1916	1	$\approx$ 1895
256	**1**	124.18	3313	1	$\approx$ 3126
1KB	**1**	56.97	7313	1	$\approx$ 7745
32KB	**1**	34.97	11983	1	$\approx$ 12488
64KB	**1**	34.62	12108	1	$\approx$ 12100

(a) $S_{req} = 128$

	Model			Experiments	
S_{2a}	w^{CPU}	w^{NET}	Max Thrp	w	Max Thrp
1KB	**1.01**	31.54	1878	1	$\approx$ 1850
2KB	**1.01**	18.64	3202	1	$\approx$ 3380
8KB	**1.03**	8.93	6791	1	$\approx$ 7050
256KB	**1.04**	5.79	10644	1	$\approx$ 10680
512KB	**1.05**	5.74	10742	1	$\approx$ 10400

(b) $S_{req} = 1$KB

	Model			Experiments	
S_{2a}	w^{CPU}	w^{NET}	Max Thrp	w	Max Thrp
8KB	**1.05**	4.92	1625	1	$\approx$ 1634
16KB	**1.08**	3.25	2530	1	$\approx$ 2687
64KB	**1.14**	2	4344	1	$\approx$ 4328
256KB	**1.17**	1.68	5293	1-2	$\approx$ 4900
512KB	**1.17**	1.63	5493	1-2	$\approx$ 4900

(c) $S_{req} = 8$KB

Table III
CLUSTER: COMPARISON OF ANALYTICAL AND EXPERIMENTAL RESULTS. PREDICTION FOR OPTIMAL w IS IN BOLD.

typical of systems where the network is comparatively faster than the process[3], which is typical on a cluster.

The model also captures the effect of batching on performance. As the size of the batches increases, the total instance time increases, as expected, reflecting the larger size, but the average time per request (ϕ_{req}^{CPU}) decreases. This is very noticeable for 128 bytes requests, with the average time per request dropping from 0.52ms down to 0.08ms for the largest batches.

Table III shows that the predicted optimal window size matches closely the value obtained in the experiments. The predicted throughput is also close to the experimental results,

[3]The speed of the process depends not only on the CPU speed but also on the efficiency of the implementation.

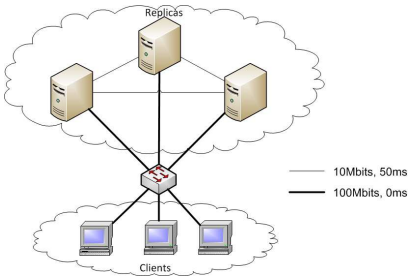


Figure 6. Topology used for Emulab experiments

with less than 5% error for 128 bytes and 1KB requests, and at most 20% for 8KB requests 512KB batch size.

B. Emulab

Figure 6 shows the topology used for the Emulab experiments, which represents a typical WAN environment with the geographically distributed nodes. We modeled this scenario with an emulated topology where the replicas are connected point-to-point by a 10Mbits link with 50ms of latency. Since the goal is to keep the system under high load, the clients are connected directly to each replica and communicate at the speed of the physical network. The physical cluster used to run the experiments consisted of nodes of Pentium III at 850MHz with 512MB of memory, connected by a 100Mbps Ethernet.

1) Experimental results: Figures 7 and 8 show results. Contrary to the cluster environment, batching alone (*i.e.*, $WND = 1$) does not suffice to achieve maximum throughput. Although larger batches improve performance significantly, batching falls short of the maximum that is achieved with larger window sizes. The difference is greater with large request sizes (1KB and 8KB), where it achieves only half of the maximum, than for small sizes (128 bytes), where batching on its own reaches almost the maximum. The reason is that with small request sizes the leader is CPU-bound, so it cannot execute more than one parallel instance, while with larger requests the bottleneck is the network latency. Increasing the window size to 2 is enough for the system to reach maximum throughput in all scenarios if the batch size is large enough (40KB with $S_{req} = 128$ and around 140KB with $S_{req} = 1$KB and $S_{req} = 8$KB). If the window size is further increased, the maximum throughput is achieved with smaller batch sizes.

The experiments also show that increasing the window size too much results in a performance collapse, with the system throughput dropping to around 10% of the maximum. This collapse happens when the leader tries to send more data than the capacity of the network, resulting in packet loss and retransmissions. The point where it happens depends on the combination of S_{req}, WND and BSZ, which indirectly control how much data is sent by the leader; the larger their values, the higher the change of performance collapse. With $S_{req} = 128$ there is no performance degradation, because the

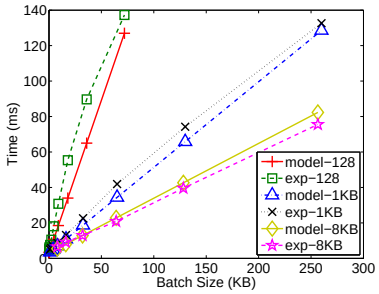


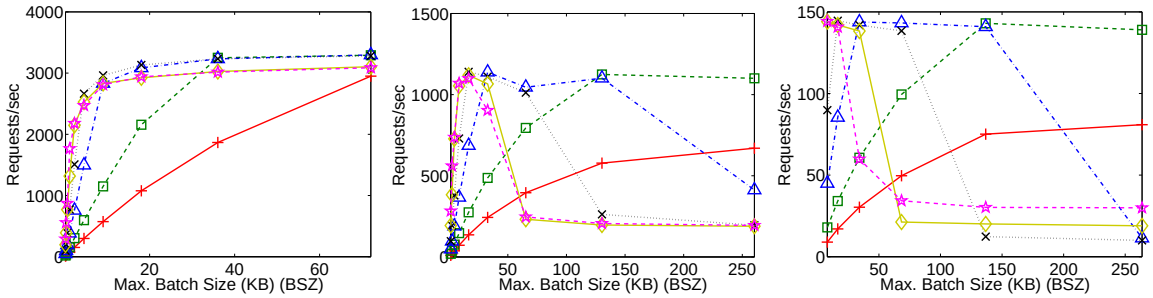
Figure 9. Experimental versus model results for the CPU time of an instance in Emulab. Fit values: $\phi_{cli}(x) = 0.28x + 0.2$, $\phi_{rep}(x) = 0.002x + 1.5$.

CPU is the bottleneck limiting the throughput. With larger request sizes, the network becomes the bottleneck and there are several cases of performance collapse. Looking at the window size, the first case occurs with $WND = 5$ with a sharp drop at $BSZ = 256$KB (Fig. 7). For larger WND, the performance collapse happens with smaller values of BSZ: with $WND = 10$ at 130KB, and at less than 64KB for larger window sizes. Similarly, as the batch size increases performance collapse occurs at smaller and smaller window sizes (Fig. 8).

These results show that CPU and the network may react to saturation very differently. In this particular system, the CPU deals gracefully with saturation (also observed in the cluster experiments), showing almost no degradation as the load increases past the point where the system reaches maximum throughput, while network saturation leads to performance collapses. The behavior may differ significantly in other implementations, because the behavior of the CPU or network when under load (graceful degradation or performance collapse) depends on the implementation of the different layers of the system, mainly application and replication framework (threading model, flow-control) but also operating system and network stack.

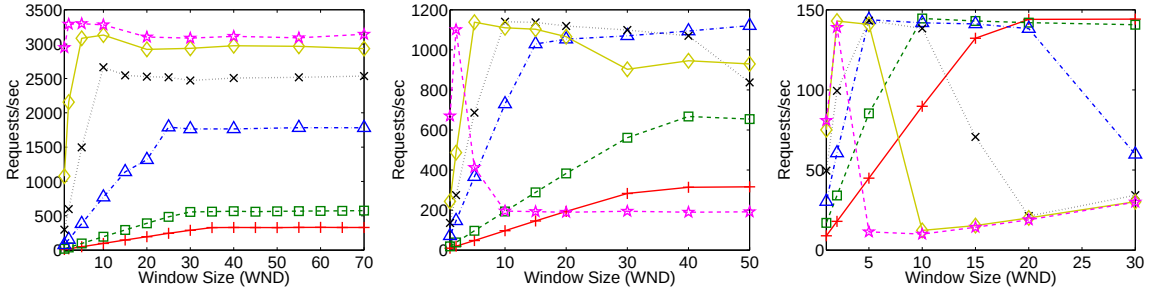
2) Setting model parameters: Following the same procedure as in the case of the cluster, we have determined the following parameters for the Emulab model: $\phi_{cli}(x) = 0.28x + 0.2$, $\phi_{rep}(x) = 0.002x + 1.5$. Figure 9 shows the training data and the corresponding model results when parametrized with the values above.

3) Comparison of analytical and experimental results: Table IV shows the results of the model for the optimal window size of the CPU and network for several batch sizes, and compares them with the experimental results. The analytical results show that the bottleneck with 128 bytes requests is the CPU (w^{CPU} is smaller than w^{NET}) while for 8KB requests it is the network. With 1KB requests, the behavior is mixed, with the CPU being the bottleneck with small batch sizes and the network with larger batch sizes. These results quantify the common sense knowledge that smaller requests and batches put a greater load on the CPU in comparison to the network. Moreover, as the request size



(a) $S_{req} = 128$ (b) $S_{req} = 1$KB (c) $S_{req} = 8$KB

Figure 7. Experimental results in Emulab: throughput with increasing batch size.



(a) $S_{req} = 128$ (b) $S_{req} = 1$KB (c) $S_{req} = 8$KB

Figure 8. Experimental results in Emulab: throughput with increasing window size.

or batch size increase, the optimal window size decreases, because if each instance contains more data, the network will be idle for less time.

The experimental results in Table IV are obtained from Figure 8. From this Figure we can determine, for each batch, the maximum throughput and the smallest w where this maximum is first achieved.

In all cases the prediction for w is in between the range where the experiments first achieve maximum throughput, showing that the model provides a good approximation. Concerning the throughput, the model is accurate with $S_{req} = 8$KB across all batch sizes. With $S_{req} = 128$, it is accurate for the smallest batches but overestimates the throughput for the larger batches. The reason is that the network can be modeled more accurately than the CPU, as it tends to behave in a more deterministic way[4]. The CPU exhibits a more non-linear behavior, especially when under high load as is the case when the number of requests in a single batch increase to more than hundreds.

V. Additional Experimental Results

Figures 10 to 18 in the appendix show the detailed experimental results for both the cluster and Emulab. For each experiment, we show six metrics: client latency, instance latency, request throughput, instance throughput, average batch size and average number of parallel instances. The *client latency* is the time the client waits for the reply to one request, which includes the transmission time from the client to the leader, the queuing time of the request at the leader, the time to order the request, and the time to send the answer back to the client.

The *latency per instance* is the time elapsed at the leader from proposal to decision of an instance, *i.e.*, from sending the Phase 2a message to receiving a majority of Phase 2b messages. It corresponds to T_{inst} (Formula 1) in the analytical model.

The *throughput of instances* is the number of Phase 2 executed per second, and the *throughput of requests* is the number of requests ordered per second. Note that the throughput of requests is equal to the throughput of instances multiplied by the average number of requests per instance. These metrics correspond to the throughput formulas of the analytical model given at the end of sections III-A1 and III-A2.

The *average batch size* (bsz) and *average window size* (w) show how well the system is taking advantage of the optimizations. As mentioned previously, the leader might not always fill the batches completely (*i.e.*, up to BSZ) or to execute the maximum number of parallel instances. This can happen either because of the lack of sufficient client requests queued for ordering or because the leader is not fast

[4]This is true only until reaching a level of saturation where packets are dropped, after which it becomes difficult to model

S_{2a}	Model w^{CPU}	Model w^{NET}	Model Max Thrp	Experiments w	Experiments Max Thrp
128	**30.88**	833.48	308	30-35	$\approx$ 330
256	**28.77**	422.94	574	25-30	$\approx$ 550
1KB	**20.45**	107.58	1620	20-25	$\approx$ 1800
16KB	**3.38**	7.68	3765	2-5	$\approx$ 3100
32KB	**1.47**	3.12	4032	1-2	$\approx$ 3300

(a) $S_{req} = 128$

S_{2a}	Model w^{CPU}	Model w^{NET}	Model Max Thrp	Experiments w	Experiments Max Thrp
1KB	**28.89**	119.01	286	30-40	$\approx$ 310
2KB	**25.54**	60.12	502	30-40	$\approx$ 600
8KB	**15.42**	15.8	1155	15-20	$\approx$ 1030
128KB	3.16	**1.93**	1184	1-2	$\approx$ 1120
256KB	2.68	**1.6**	1184	1-2	$\approx$ 1100

(b) $S_{req} = 1KB$

S_{2a}	Model w^{CPU}	Model w^{NET}	Model Max Thrp	Experiments w	Experiments Max Thrp
8KB	19.47	**16**	150	15-20	$\approx$ 144
16KB	14.24	**8.5**	150	5-10	$\approx$ 144
64KB	6.72	**2.88**	150	2-5	$\approx$ 144
128KB	4.84	**1.94**	150	1-2	$\approx$ 144
256KB	3.8	**1.47**	150	1-2	$\approx$ 144

(c) $S_{req} = 8KB$

Table IV
EMULAB: COMPARISON OF ANALYTICAL AND EXPERIMENTAL RESULTS. PREDICTION OF OPTIMAL w IS IN BOLD.

enough to execute WND parallel instances simultaneously (previous instances finish before the leader is able to start additional ones). Therefore, we measured the average size of the batches and the average number of parallel instances and show the results below, in order to evaluate the effectiveness of the optimizations in each scenario. Formula (8) of the analytical model gives an estimation for the average window size for the case when the batches are full (*i.e.*, $bsz = BSZ$), and can be compared with the results below.

For the cluster experiments, we only show the plots with the metrics as a function of maximum batch size (Figures 10 to 12), while for the emulab experiments we show them both as a function of maximum batch size (Figures 13 to 15) and of maximum window size (Figures 16 to 18). The reason is that in the cluster increasing the window size does not affect performance significantly, so the corresponding plots would not provide any additional information.

A. General comments

Client versus instance latency: In most experiments, the client latency is generally one or two orders of magnitude higher than the instance latency: while the instance latency is usually below 5ms in the cluster and 500ms in Emulab, the client latency is usually above 100ms and 1s, respectively. This is explained by the high load created by the clients, which surpasses by far the capacity of the system. The requests received by the leader are first put on a waiting queue until the leader is ready to propose it. This might take a long time since the leader is limited in how many requests it can propose by the bounds WND and BSZ, as well as by its speed. This queuing time before proposal is what explains the difference between client and instance latency. In most cases, increasing BSZ and WND improves the request throughput, which in turn decreases the client latency (see for instance, Figures 13a and 13d). This happens in spite of the instance latency also increasing (Figure 13b), because when the system is under heavy load, the instance latency is just a small part of the total latency experienced by the clients, with the queuing delays being the major cause of delay.

Instance versus request throughput: A common pattern is that larger batch sizes result in lower instance throughput but higher request throughput (See, as an example, Figures 13c, 13e and 13d). A larger batch size increases the number of requests ordered per instance, which potentially improves request throughput. On the other hand, larger batches increase the time necessary to order an instance, which decreases the instance throughput, potentially decreasing request throughput. But as the results show, the gains from the additional requests in each batch are greater than the losses from the lower instance throughput, so overall there is a net improvement in request throughput.

B. Cluster

Effectiveness of batching: Figures 10c, 11c and 12c show that the leader is always able to fill up the batches to their maximum size, thereby taking full advantage of the batching optimization. This increase in batch size translates initially into an improvement in request throughput, and then a stabilization when the CPU is saturated. After this point, although the batches are still full, the instance time increases, canceling out the gains in batch size.

Effectiveness of parallel instances: On the other hand, increasing WND does not improve throughput in the cluster because the leader is CPU bound. The plots with the average number of instances (w) (Figures 10f, 11f and 12f) show what happens: for request sizes of 128 and 1KB, the leader is not able to execute more than one instance in parallel, as it does not have time to start more instances before the previous one finish. For request sizes of 8KB, the leader executes an average of two instances in parallel, but the gains here are offset by a corresponding increase in instance latency (Fig. 12b), and therefore have no impact on overall throughput.

Instance latency: The instance latency (Fig. 10b, 11b and 12b) increases linearly with the batch size, since the larger batch size requires longer to process and transmit. The only exception is with small batch sizes (< 512 bytes), where the curves for $WND = 2$ and $WND = 5$ peak at around 4ms as compared to 0.4ms for larger batch sizes. This additional overhead is because the system is executing multiple instances in parallel (Fig. 10f, 11f and 12f). Notice that with $WND = 1$ this peak does not occur. For

larger values of BSZ, the system is only able to execute one instance at a time, and therefore the instance latency drops.

Behavior under saturation: As BSZ and WND increase, the system reaches the maximum capacity, which corresponds to the saturation of the leader's CPU. We have concluded this by measuring the average CPU utilization of the leader, which is well over 90% in the experiments where the system is at maximum capacity. However, we can reach the same conclusion by observing that the network is not saturated. Considering requests of 8KB, the maximum throughput of 5000 req/sec corresponds to around 40MB/sec of requests ordered, which translate to a total network traffic on the outgoing link of the leader of 80MB/sec, as each request has to be sent to the two other replicas. This is below the maximum data rate supported by the cluster, which is approximately 117.5MB/sec (=940Mbits). For 1KB request sizes, the total data sent on the outgoing link of the leader peaks at around 20MB, and for requests of 128 bytes at 3MB, so the network is mostly idle.

An important observation is that the system behaves gracefully after reaching maximum capacity, with no observable drop in performance. This is highly desirable, as it simplifies choosing values for BSZ and WND. Nevertheless, these results should not be used to conclude that setting a very high value for these parameters is safe. For instance, increasing the batch size too much may introduce delays if the arrival rate of client requests is not high enough to fill the batches quickly.

C. Emulab

Figures 13, 14 and 15 show the experimental results in Emulab as a function of maximum batch size, while Figures 16, 17 and 18 show the same metrics as a function of maximum window size.

Effectiveness of batching: Batching by itself (Series $WND = 1$ in Figures 13, 14 and 15) increases the throughput significantly but, contrary to the cluster environment, it falls short of the maximum throughput, which is reached by a combination of batching and parallel instances. With request size of 1KB and 8KB, the best that is reached with batching alone is about half of the maximum throughput and the shape of the curve of $WND = 1$ suggests that further increasing the batch size would not lead to any more significant increases in throughput. With a request size of 128 bytes a batch size of around 70KB is almost enough to reach the maximum of 3200 requests/sec, and the suggests that further increasing the batching would reach the maximum. But even in this case, batching by itself might be a bad choice, as filling up a batch of 70KB with requests of 128 bytes, requires a little over 500 client requests, which might not be a practical number.

With requests sizes of 128 bytes and 1KB, the leader is not able to completely fill the batches if $WND > 1$ (Figures 13c and 14c). In these cases, the average batch size stabilizes at around 30KB and 100KB, respectively, even as BSZ is increase well past these values. This is caused by insufficient client requests to fill the batch completely within the batching delay, which is set to 50ms (see description of batching algorithm in Section IV).

Effectiveness of parallel instances: The parallel instance optimization is not enough by itself to reach the maximum throughput of the system with request sizes of 128 bytes and 1KB, but with 8KB requests it does reach the maximum throughput when $WND > 15$ (Series $BSZ = 128$, $BSZ = 1KB$, and $BSZ = 8KB$ of Figures 16d, 17d and 18d).

With request sizes of 128 bytes the improvement in throughput from this optimization alone is modest. Although the leader is always able to execute up to WND parallel instances (Figure 16f), with $WND > 35$ the instance latency starts increasing (Figure 16b), leading to a stabilization in request throughput. This shows one of the limitations of the parallel instances optimization, which imposes additional overhead on the CPU of the leader and limits its effectiveness to the cases where there are spare CPU resources. When the CPU becomes saturated, the leader takes longer to process the messages received which increases the instance latency.

Combining batching and parallel instances: The highest throughput is reached when batching and parallel instances are used in combination. There are several optimal combinations of BSZ and WND, with the optimal value of one of the parameters being inversely proportional to the optimal of the other. As an example, with 128 byte requests (Figure 13d), when $WND = 2$ the highest throughput is reached with $BSZ = 40$, but when $WND = 10$ a maximum batch size of 10KB is already enough.

Network congestion: As mentioned previously, for request sizes of 1KB and 8KB, the performance collapses when BSZ and WND are increases past certain values due to network saturation. This is clearly visible in Figures 14b and 15b, which show that as the request throughput collapses, the instance latency increases substantially. The increased latency is the result of packet loss, whose effect is particularly significant in high latency network, as is the case in these experiments.

VI. Discussion

The experiments show clearly that batching by itself provides the largest gains both in high and low latency networks. Since it is fairly simple to implement, it should be one of the first optimizations considered in Paxos and, more generally, in any implementation of a replicated state machine.

Pipelining is useful only in some systems, as its potential for throughput gains depends on the ratio between the speed of the nodes and the network latency: the more time the leader spends idle waiting for messages from other replicas,

the greater the potential for gains of executing instances in parallel. Thus, in general, it will provide minimal performance gains over batching alone in low latency networks, but it provides substantial gains when latency is high.

While batching decreases the CPU overhead of the replication stack, executing parallel instances has the opposite effect because of the overhead associated with switching between many small tasks. This reduces the CPU time available for the service running on top of the replication task and, in the worst case, can lead to a performance collapse if too many instances are started simultaneously (see Emulab experiments). This problem can be avoided by carefully setting the limit on the number of parallel instances, taking in consideration the available CPU time on the leader. The analytical model in this paper helps in choosing this value, by providing the minimal window size that results in optimal throughput for a given batch size.

The paper has focused on throughput because as long as latency is kept within an acceptable range, optimizing throughput provides greater gains in overall performance. A system tuned for high-throughput will have higher capacity, therefore being able to serve a higher number of clients with an acceptable latency, whereas a system tuned for latency will usually reach congestion with fewer clients, at which point its performance risks collapsing to values well below the optimal.

VII. Related Work

The two optimizations to Paxos studied in this paper are particular cases of general techniques widely used in distributed systems. Batching is an example of message aggregation, which has been previously studied as a way of reducing the fixed per-packet overhead by spreading it over a large number of data or messages, see [3]–[6]. It is also widely deployed, with TCP's Nagle algorithm [10] being a notable example. Pipelining is a general optimization technique, where several requests are executed in parallel to improve the utilization of resources that are only partially used by each request. One of the main examples of this technique is HTTP pipelining [11]. The work in this paper looks at these two optimizations in the context of state machine replication protocols, studying how to adapt them and combine them in Paxos. Most implementations of replicated state machines use batching and pipelining to improve performance, but as far as we are aware, there is no detailed study on combining these two optimizations.

In [3], the authors use simulations to study the impact of batching on several group communication protocols. The authors conclude that batching provides one to two orders of magnitude gains both on latency and throughput. A more recent work [5] proposes an adaptive batching policy also for group communication systems. In both cases the authors look only at batching. In this paper, we have shown that pipelining should also be considered, as in some scenarios batching by itself is not enough for optimal performance.

Batching has been studied as a general technique by [4] and [6]. In [4] the authors present a detailed analytical study, quantifying the effects of batching on reliable message transmission protocols. One of the main difficulties in batching is deciding when to stop waiting for additional data and form a batch. This problem was studied in [6], where the authors propose two adaptive batching policies. The techniques proposed in these papers can easily be adapted to improve the batching policy used in our work, which was kept simple on purpose as it was not our main focus.

There are a few experimental studies showing the gains of batching in replicated state machines. One such example is [12], which describes an implementation of Paxos that uses batching to minimize the overhead of stable storage.

Batching is especially important in Byzantine systems, as these protocols are more expensive than the corresponding protocols for benign faults due to a higher message complexity and the use of cryptographic operations. Two examples are PBFT [13] and Zyzzyva [14], both of which use batching and pipelining. The corresponding publications contain experimental studies that, among other factors, evaluate the effects of batching. But these studies have a limited scope, focusing only on a narrow range of settings and ignoring the interplay with pipelining.

There has been a lot of work on other optimizations for improving the performance of Paxos-based protocols. LCR [15] is an atomic broadcast protocol based on a ring topology and vector clocks that is optimized for high throughput. Ring Paxos [7] is a variant of the Paxos protocol, that combines several techniques, like IP multicast, ring topology, and using a minimal quorum of acceptors, to maximize network utilization. These two papers consider only a LAN environment and, therefore, use techniques that are only available on a LAN (IP multicast) or that are effective only if network latency is low (ring-like organization). We make no such assumptions in our work, so our work applies both to WAN and LAN environments. In particular, pipelining is a especially effective technique in medium to high-latency networks, so it is important to understand its behavior.

VIII. Conclusion

In this paper we have studied two important optimizations to Paxos, batching and pipelining. The analytical model presented in the paper is effective at predicting the combinations of batch size and number of parallel instances that result in optimal throughput in a given system, and therefore can be used to assist in tuning a Paxos deployment for maximum throughput.

Additionally, we have shown that batching produces the largest gains, both in a cluster and a WAN environment. Together with its simplicity, these results suggest that batching should be the first optimization considered in such a system.

Interestingly, in systems with moderate to high network latency, batching by itself is no longer enough to achieve the best throughput. In this case, the use of pipelining provides a significant improvement in performance. The results show that as the network latency increases, the gains of pipelining become more significant.

ACKNOWLEDGMENT

The authors would like to thank Paweł T. Wojciechowski, Jan Kończak and Tomasz Żurkowski for their work on JPaxos.

REFERENCES

[1] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, May 1998.

[2] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, Dec. 2004.

[3] R. Friedman and R. Renesse, "Packing messages as a tool for boosting the performance of total ordering protocols," Department of Computer Science, Cornell University, Tech. Rep. TR95-1527, 1995.

[4] B. Carmeli, G. Gershinsky, A. Harpaz, N. Naaman, H. Nelken, J. Satran, and P. Vortman, "High throughput reliable message dissemination," in *Proceedings of the 2004 ACM Symposium on Applied Computing*, NY, USA, 2004.

[5] A. Bartoli, C. Calabrese, M. Prica, E. Di Muro, and A. Montresor, "Adaptive message packing for group communication systems," in *OTM 2003 Workshops*, ser. LNCS. Springer, 2003.

[6] R. Friedman and E. Hadad, "Adaptive batching for replicated servers," in *Symposium on Reliable Distributed Systems, SRDS'06*, Oct. 2006.

[7] P. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring paxos: A high-throughput atomic broadcast protocol," in *Dependable Systems and Networks (DSN'10)*, Jun. 2010.

[8] B. White and J. L. et al, "An integrated experimental environment for distributed systems and networks," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[9] N. Santos, J. Konczak, T. Zurkowski, P. Wojciechowski, and A. Schiper, "Jpaxos - state machine replication in java," EPFL, Tech. Rep. to appear, 2011.

[10] J. Nagle, "Congestion control in ip/tcp internetworks," IETF, Tech. Rep. RFC 896, Jan. 1984.

[11] V. N. Padmanabhan and J. C. Mogul, "Improving http latency," *Computer Networks and ISDN Systems*, vol. 28, no. 1-2, 1995.

[12] Y. Amir and J. Kirsch, "Paxos for system builders," Johns Hopkins University, Tech. Rep. CNDS-2008-2, 2008.

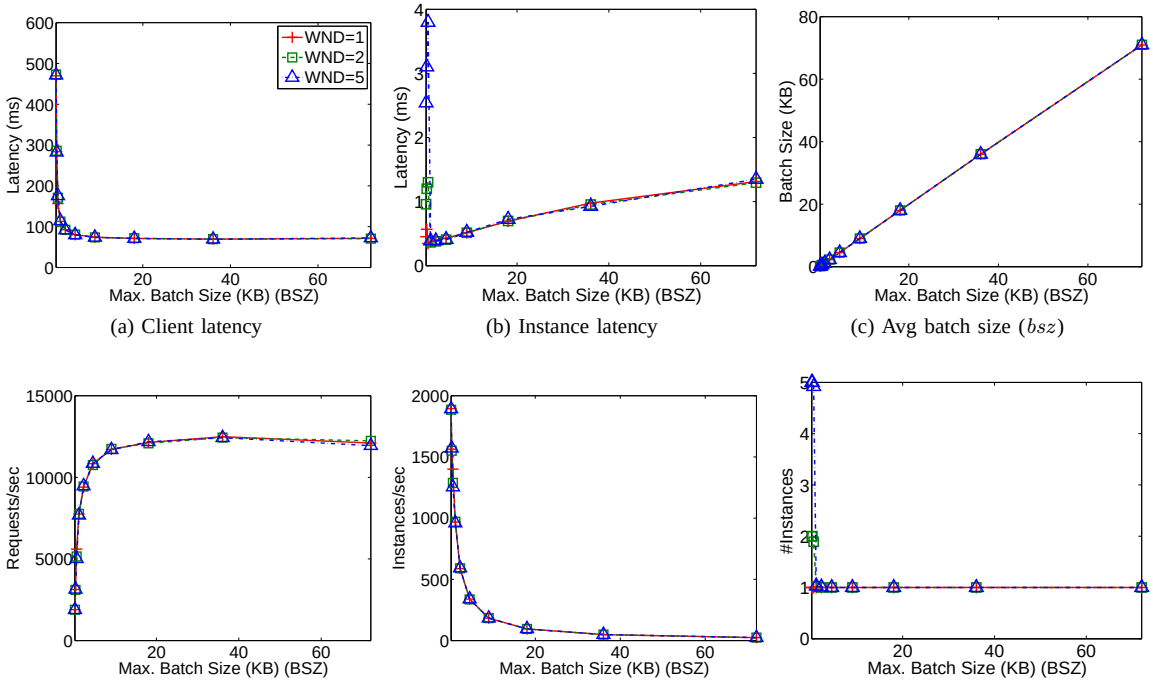
[13] M. Castro, "Practical byzantine fault tolerance," Ph.D. dissertation, Laboratory for Computer Science, MIT, 2001.

[14] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of twenty-first ACM SIGOPS SOSP*, NY, USA, 2007.

[15] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, "Throughput optimal total order broadcast for cluster environments," *ACM Trans. Comput. Syst.*, vol. 28, no. 2, 2010.

APPENDIX

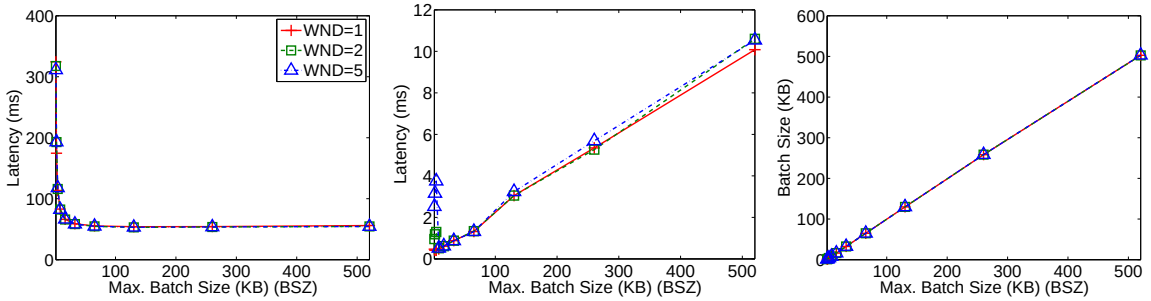
A. Cluster: additional experimental results



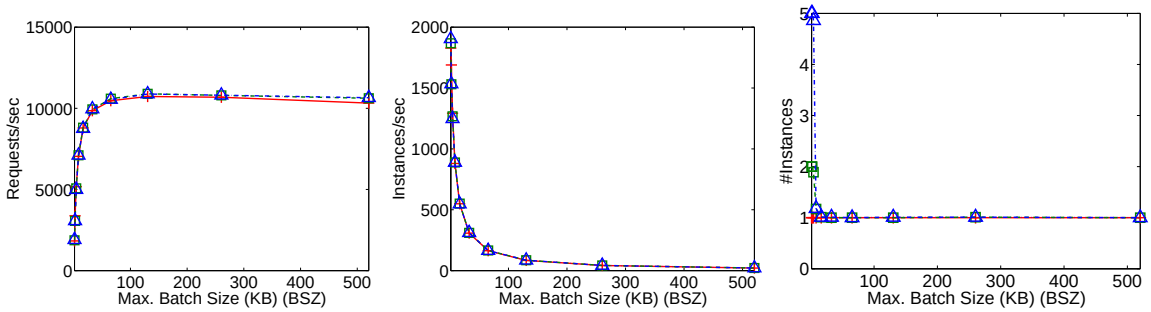
(a) Client latency (b) Instance latency (c) Avg batch size (bsz)

(d) Requests/sec (e) Instances/sec (f) Avg window size (w)

Figure 10. Cluster, experimental results with $S_{req} = 128$.

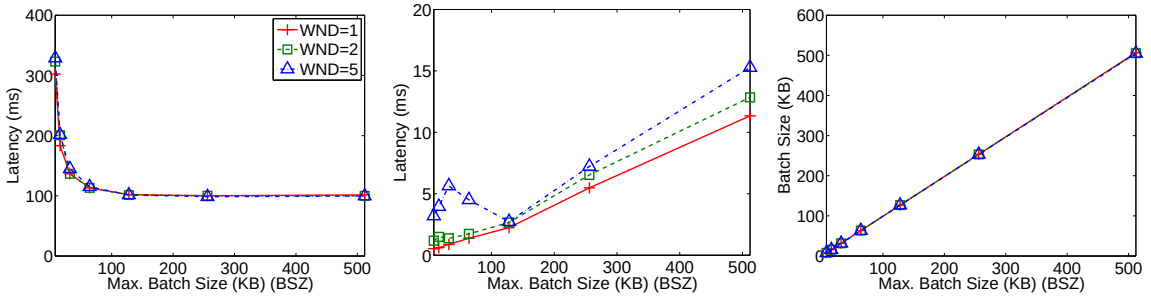


(a) Client latency (b) Instance latency (c) Avg batch size (bsz)

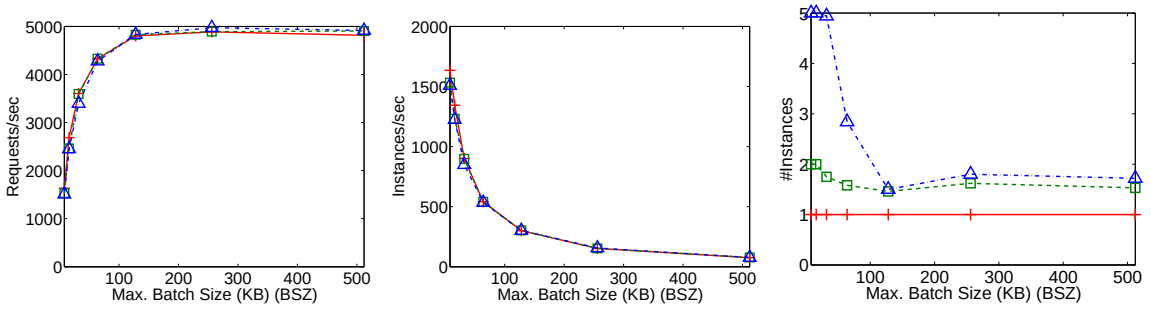


(d) Requests/sec (e) Instances/sec (f) Avg window size (w)

Figure 11. Cluster, experimental results with $S_{req} = 1$KB.



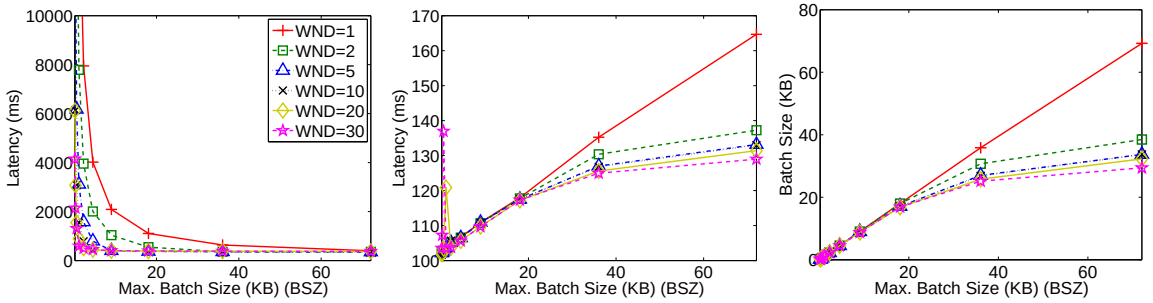
(a) Client latency (b) Instance latency (c) Avg batch size (bsz)



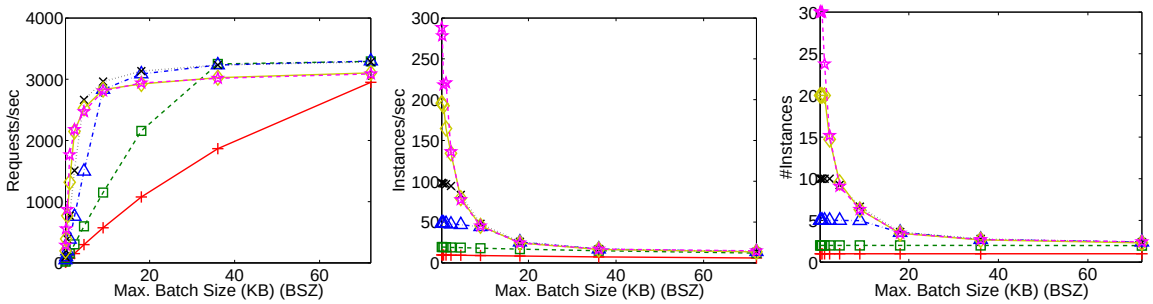
(d) Requests/sec (e) Instances/sec (f) Avg window size (w)

Figure 12. Cluster, experimental results with $S_{req} = 8$KB.

B. *Emulab: additional experimental results*

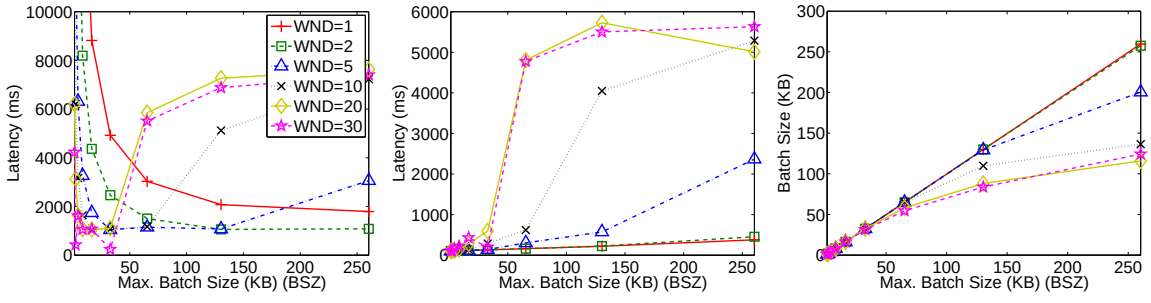


(a) Client latency (b) Instance latency (c) Avg batch size (bsz)

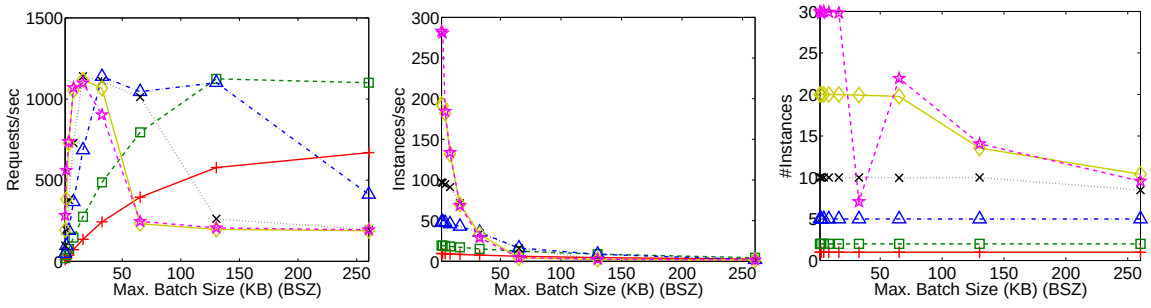


(d) Requests/sec (e) Instances/sec (f) Avg window size (w)

Figure 13. Emulab, experimental results with $S_{req} = 128$.

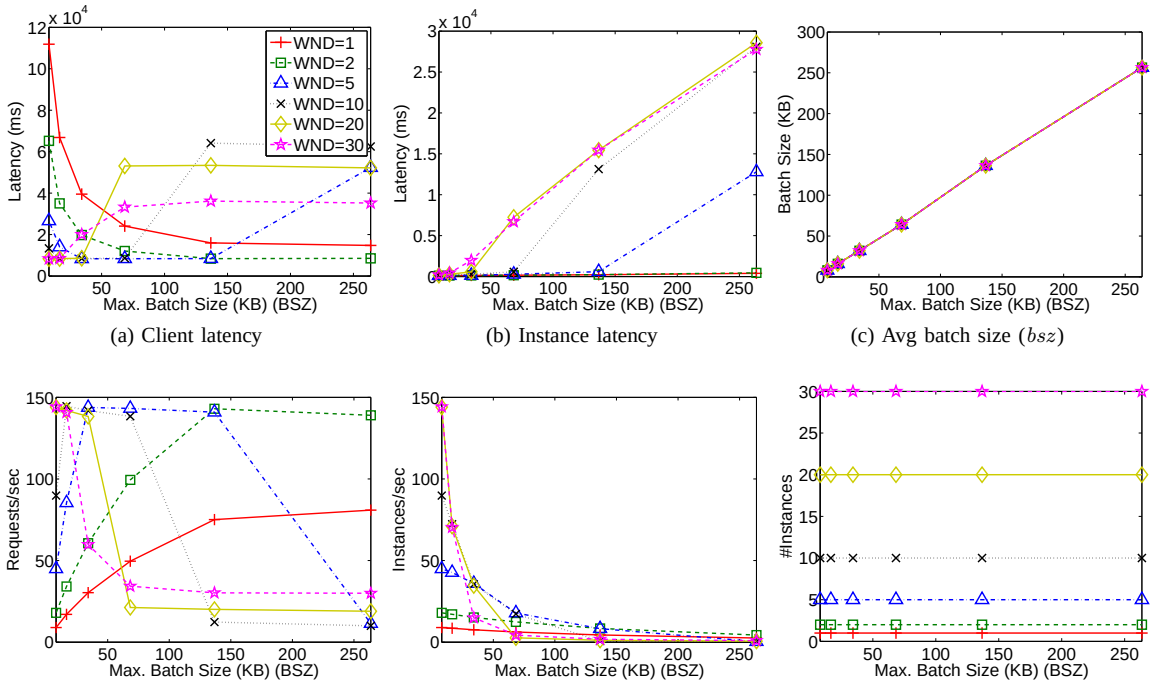


(a) Client latency (b) Instance latency (c) Avg batch size (bsz)



(d) Requests/sec (e) Instances/sec (f) Avg window size (w)

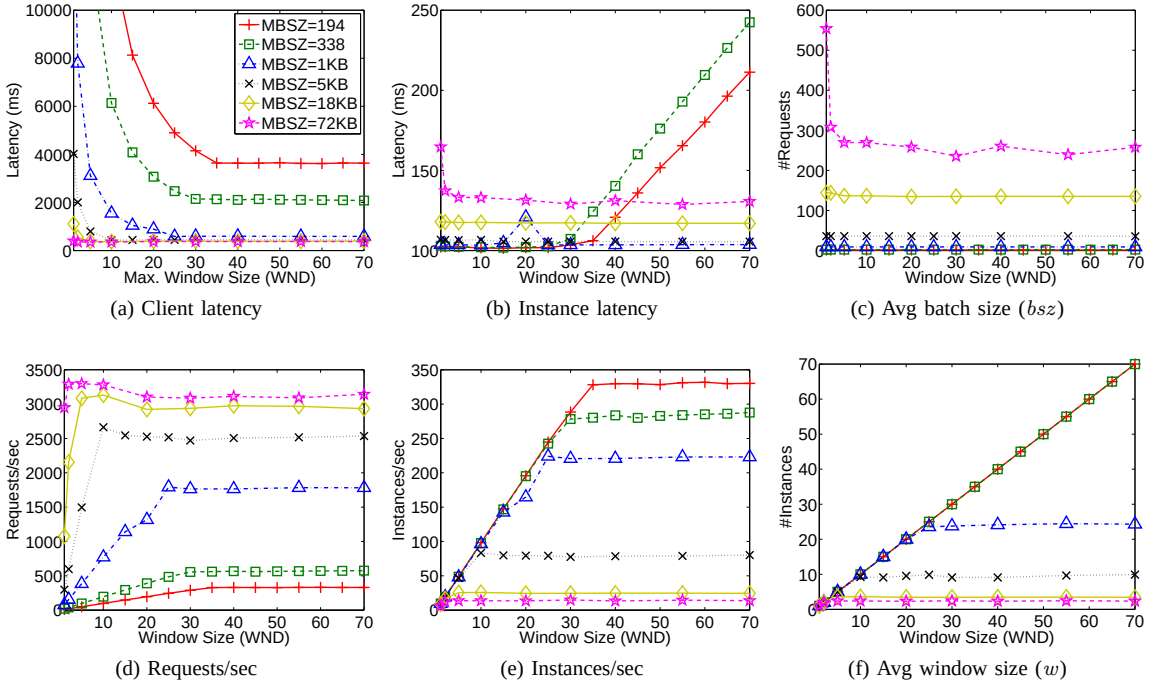
Figure 14. Emulab, experimental results with $S_{req} = 1$KB.



(a) Client latency (b) Instance latency (c) Avg batch size (bsz)

(d) Requests/sec (e) Instances/sec (f) Avg window size (w)

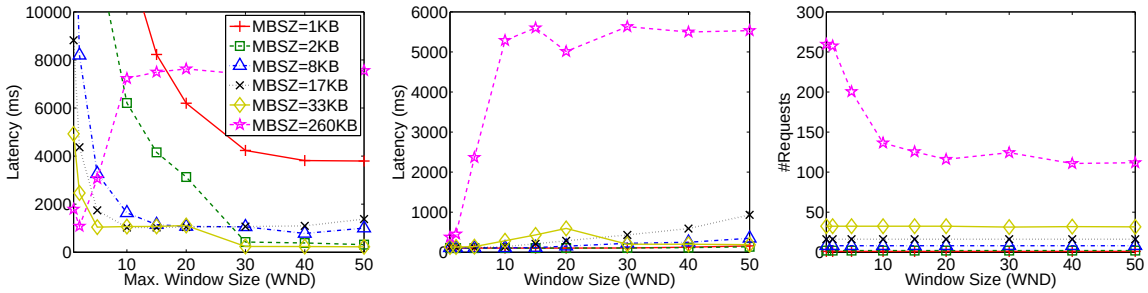
Figure 15. Emulab, experimental results with $S_{req} = 8$KB.



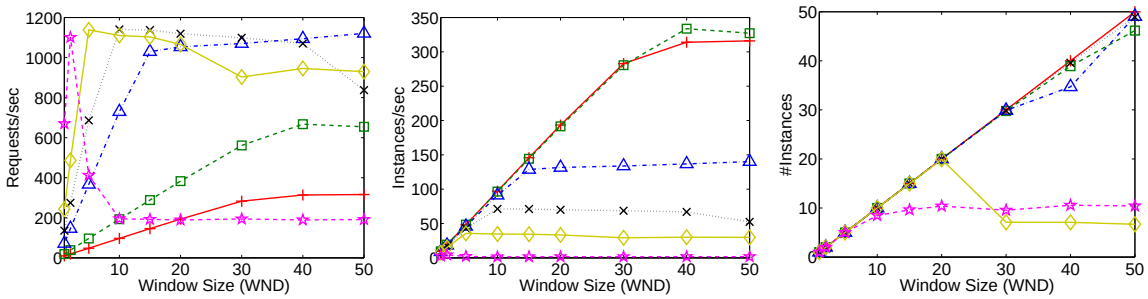
(a) Client latency (b) Instance latency (c) Avg batch size (bsz)

(d) Requests/sec (e) Instances/sec (f) Avg window size (w)

Figure 16. Emulab, experimental results with $S_{req} = 128$.

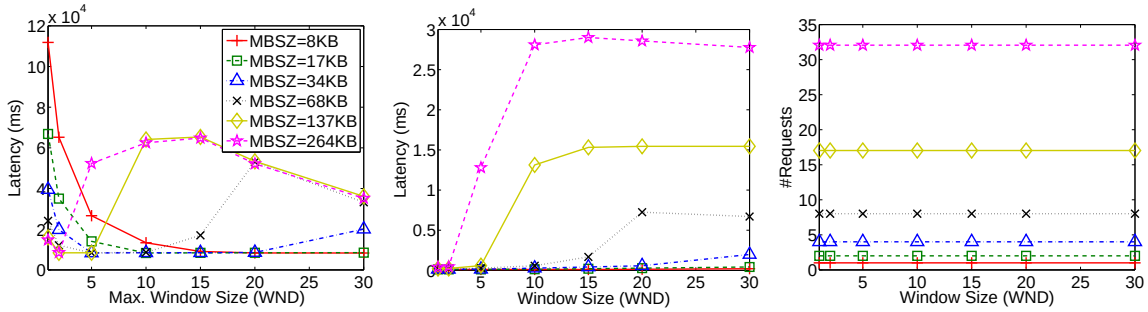


(a) Client latency (b) Instance latency (c) Avg batch size (bsz)

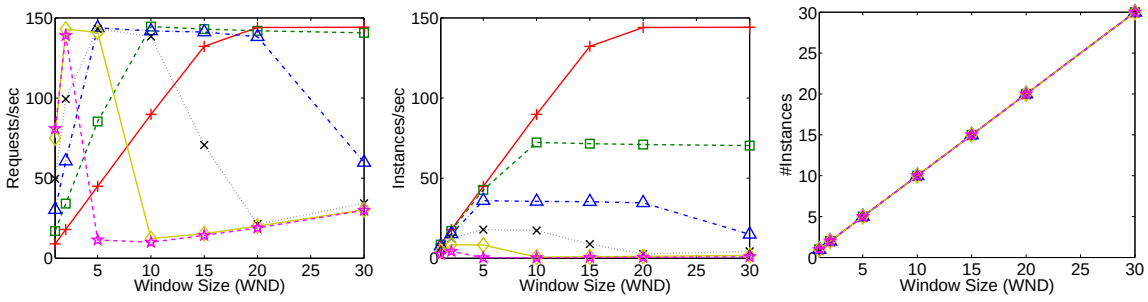


(d) Requests/sec (e) Instances/sec (f) Avg window size (w)

Figure 17. Emulab, experimental results with $S_{req} = 1$KB.



(a) Client latency (b) Instance latency (c) Avg batch size (bsz)



(d) Requests/sec (e) Instances/sec (f) Avg window size (w)

Figure 18. Emulab: experimental results with $S_{req} = 8$KB.